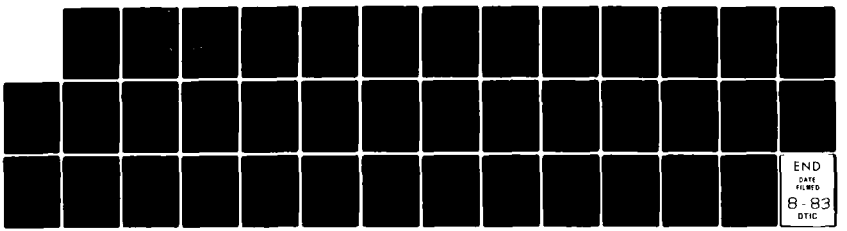
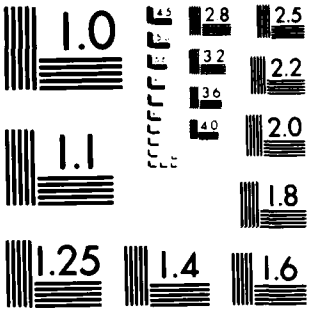
AD-A130 625 A MECHANICAL PROOF OF THE TURING COMPLETENESS OF PURE LISP(U) TEXAS UNIV AT AUSTIN INST FOR COMPUTING SCIENCE AND COMPUTER A.. R S BOYER ET AL. MAY 83 ICSCA-CMP-37 1/1

UNCLASSIFIED N00014-81-K-0634 F/G 9/2 NL





SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ICSCA-CMP-37	2. GOVT ACCESSION NO.: AD-A130625
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): A MECHANICAL PROOF OF THE TURING COMPLETENESS OF PURE LISP	5. TYPE OF REPORT & PERIOD COVERED: Technical
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Robert S. Boyer & J Strother Moore	8. CONTRACT OR GRANT NUMBER(s): MCS-8202943; N00014-81-K-0634
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Institute for Computing Science & Computer Applications, The University of Texas at Austin, Main Building 2100, Austin, Texas 78712	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: NR 049-500
11. CONTROLLING OFFICE NAME AND ADDRESS: Software Systems Science, National Science Found., Washington, DC 20550; Office of Naval Research, 800 N. Quincy St, Arlington, VA 22217	12. REPORT DATE: May, 1983 13. NUMBER OF PAGES: 39 pages
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Reproduction in whole or in part is permitted for any purposes of the United States government.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Automatic theorem-proving, computability, interpreters, LISP, program verification, recursive unsolvability, termination, Turing machines.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The authors describe a proof by a computer program of the Turing completeness of a computational paradigm akin to Pure LISP. That is, they define formally the notions of a Turing machine and of a version of Pure LISP and prove that anything that can be computed by a Turing machine can be computed by LISP. While this result is straightforward, they believe this is the first instance of a machine proving the Turing completeness of another computational paradigm.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Table of Contents

Abstract

We describe a proof by a computer program of the Turing completeness of a computational paradigm akin to Pure LISP. That is, we define formally the notions of a Turing machine and of a version of Pure LISP and prove that anything that can be computed by a Turing machine can be computed by LISP. While this result is straightforward, we believe this is the first instance of a machine proving the Turing completeness of another computational paradigm.

Key words: automatic theorem-proving, computability, interpreters, LISP, program verification, recursive unsolvability, termination, Turing machines.

1. Introduction

In our paper [3] we present a definition of a function EVAL that serves as an interpreter for a language akin to Pure LISP, and we describe a mechanical proof of the unsolvability of the halting problem for this version of LISP. We claim in that paper that we have proved the recursive unsolvability of the halting problem. It has been pointed out by a reviewer that we cannot claim to have mechanically proved the recursive unsolvability of the halting problem unless we also mechanically prove that the computational paradigm used is as powerful as partial recursive functions. To this end we here describe a mechanically checked proof that EVAL is at least as powerful as the class of Turing machines. The proof was constructed by the same theorem-proving system used to prove the unsolvability result, a descendant of the systems described in [1] and [2].

One of our motivations in writing this paper is to add further support to our conjecture that, with the aid of automatic theorem-provers, it is now often practical to do mathematics formally. Our proof of the completeness theorem is presented as a sequence of lemmas, stitched together by some English sentences motivating or justifying the lemmas. Our description of the proof is at about the same level it would be in a careful informal

presentation of Turing completeness. However, every formula exhibited as a theorem in this document has actually been proved mechanically. In essence, our machine has read the informal proof and assented to each step (without bothering with the motivations). Thus, while the user of the machine was describing an informal proof at a fairly high level, the machine was filling in the often large gaps.

In the next three sections we informally describe our formalization of LISP and of Turing machines, and we develop a formal statement of the Turing completeness of LISP. The rest of the document is an extensively annotated list of the commands used to lead our theorem-prover to the proof of the completeness result. We assume throughout that the reader is familiar with our formal logic as described in [1]. We also assume the reader is somewhat familiar both with the LISP programming language and a classical treatment of Turing machines, such as that by Rogers [4].

2. Pure LISP

Suppose X is a LISP s-expression, VA is an association list ("alist") binding LISP variable names to values, and FA is an alist binding LISP function names to function definitions. The ideal LISP interpreter takes three arguments, X, VA, and FA, and either returns the value of X under VA and FA, or runs forever.

Since we wish to prove theorems about LISP programs we would like to define in our logic the ideal LISP interpreter. However, our definitional principle requires that when a new definitional axiom of the form eval(X,VA,FA) = body is proposed, there exist a measure of <X,VA,FA> that decreases in a well-founded sense in every call of eval in body. This prevents us from directly defining the ideal interpreter as a function. Instead we define the function EVAL which takes four arguments, X, VA, FA, and N. N can be thought of as the amount of stack space available to the interpreter. (EVAL X VA FA N) returns

either the value of X under VA and FA, or else it returns the special object (BTM) to denote that it ran out of stack space.

(BTM) is an object of a "new" shell type and is not equal to T, F, any number, literal atom, or list. (BTM) is recognized by the function BTMP, which returns T or F according to whether its argument is equal to (BTM).

We now briefly illustrate EVAL. Let va be an alist in which A has the value '(1 2 3), A0 has the value '(1 2 3 . 0), and B has the value '(A B C D).

```
va = '((A . (1 2 3))
       (A0 . (1 2 3 . 0))
       (B . (A B C D))).
```

Let fa be an alist in which the atom 'APP is defined as shown below:

```
fa = '((APP (X Y)
            (IF (EQUAL X NIL)
                Y
                (CONS (CAR X) (APP (CDR X) Y)))))
```

Then here are two theorems about EVAL:

```
(EVAL '(APP A B) va fa N) = (IF (LESSP N 4)
                                (BTM)
                                '(1 2 3 A B C D))

(EVAL '(APP A0 B) va fa N) = (BTM)
```

The proof of the second theorem depends on the fact that the CDR of 0 is 0, which is not NIL.

The informal notion that the computation of X under VA and FA "terminates" is formalized by "there exists a stack depth N such that (EVAL X VA FA N) ≠ (BTM)." Conversely, the notion that the computation "runs forever" is formalized by "for all N (EVAL X VA FA N) = (BTM)." Thus, we see that the computation of (APP A B) under va and fa above terminates, while the computation of (APP A0 B) under va and fa does not. Since our logic does not contain quantifiers, we will have to derive entirely constructive replacements

of these formulas to formulate the desired theorem.

The reader should see [3] for our formal definition of EVAL.

3. Turing Machines

Our formalization of Turing machines follows the description by Rogers [4]. The ideal Turing machine interpreter takes three arguments: a state symbol, ST, an infinite tape TAPE (with a distinguished "cell" marking the initial position of the Turing machine), and a Turing machine TM. If the computation terminates, the Turing machine interpreter returns the final tape. Otherwise, the ideal interpreter runs forever.

We cannot define the ideal interpreter directly as a function in our logic. Instead, we define the function TMI of four arguments, ST, TAPE, TM, and K. The fourth argument is the maximum number of state transitions permitted. TMI returns either the final tape (if the machine halts after executing fewer than K instructions) or (BTM). We say the computation of TM (starting in state ST) on TAPE "terminates" if "there is a K such that (TMI ST TAPE TM K) $\neq$ (BTM)" and "runs forever" if "for all K (TMI ST TAPE TM K) = (BTM)."

The primary difference between our formalization of Turing machines and Rogers' description is that we limit our attention to infinite tapes containing only a finite number of 1's. This is acceptable since Rogers uses only such tapes to compute the partial recursive functions (page 14 of [4]):

> Given a Turing machine, a tape, a cell on that tape, and an initial internal state, the Turing machine carries out a uniquely determined succession of operations, which may or may not terminate in a finite number of steps. We can associate a partial function with each Turing machine in the following way. To represent an input integer x, take a string of x+1 consecutive 1's on the tape. Start the machine in state q0 on the leftmost cell containing a 1. As output integer, take the total number of 1's appearing anywhere on the tape when (and if) the machine stops.

Our conventions for formalizing states, tapes, and Turing machines are made

clear in Section 5.

4. The Statement of the Turing Completeness of LISP

We wish to write down a formula that expresses the idea that LISP can compute anything a Turing machine can compute. Roughly speaking we wish to establish a correspondence between Turing machines, their inputs, and their outputs on the one hand, and LISP programs, their inputs, and their outputs on the other. Then we wish to prove (a) that if a given Turing machine runs forever on given input then so does its LISP counterpart; and (b) if a given Turing machine terminates on given input then its LISP counterpart terminates with the same answer.

We will set up a correspondence between Turing machines and function alists (since it is there that LISP programs are defined). The mapping will be done by the logical function tmi.fa, which takes a Turing machine and returns a corresponding LISP program. We will also set up a correspondence between the state and tape "inputs" of the Turing machine and the s-expression being evaluated by LISP, since the s-expression can be thought of as the "input" to the LISP program defined on the function alist. That mapping is done by the function tmi.x. In our statement of the completeness theorem, we use the NIL variable alist.

Since the value of a terminating Turing machine computation is a tape, and tapes are formalized as list structures that can be manipulated by LISP programs, we will require that the corresponding LISP computation deliver the list structure identical to the final tape.

Our informal characterization of the Turing completeness of LISP is then as follows. Let TM be a Turing machine, ST a state, and TAPE a tape. Let fa_{TM} be the corresponding LISP program (i.e., the result of (tmi.fa TM)), and let $x_{ST,TAPE}$ be the corresponding LISP expression (i.e., the result of (tmi.x ST

TAPE)). Then

(a) If the Turing machine computation of TM (starting in ST) on TAPE runs forever, then so does the LISP computation of $x_{ST,TAPE}$ under the NIL variable alist and the function alist fa_{TM}.

(b) If the Turing machine computation of TM (starting in ST) on TAPE terminates with tape t, then the LISP computation of $x_{ST,TAPE}$ under NIL and fa_{TM} terminates with result t.

To formalize (a) we replace the informal notions of "runs forever" by the formal ones. The result is:

all K (TMI ST TAPE TM K)=(BTM),
  ->
all N (EVAL $x_{ST,TAPE}$ NIL fa_{TM} N)=(BTM).

This is equivalent to:

exists N (EVAL $x_{ST,TAPE}$ NIL fa_{TM} N) ≠ (BTM),
  ->
exists K (TMI ST TAPE TM K) ≠ (BTM).

The existential quantification on N in the hypothesis can be transformed into implicit universal quantification on the outside. The existential quantification on K in the conclusion can be removed by replacing K with a term that delivers a suitable K as a function of all the variables whose (implicit) quantifiers govern the occurrence of K. The result is:

(a') (EVAL $x_{ST,TAPE}$ NIL fa_{TM} N)≠ (BTM)
  ->
(TMI ST TAPE TM (tmi.k ST TAPE TM N))≠ (BTM).

That is, we prove that a suitable K exists by constructively exhibiting one.

To formalize part (b) observe that it is equivalent to

If the computation of TM (starting in ST) on TAPE terminates within K steps with tape t, then there exists an N such that t = (EVAL $x_{ST,TAPE}$ NIL fa_{TM} N).

or, equivalently,

(TMI ST TAPE TM K) ≠ (BTM),
->
exists N (TMI ST TAPE TM K) = (EVAL $x_{ST,TAPE}$ NIL fa_{TM} N)

We remove the existential quantification on N by replacing N with a particular term, (tmi.n ST TAPE TM K), that delivers an appropriate N as a function of the other variables.

Our formal statement of the Turing completeness of LISP is:

```
Theorem.  TURING.COMPLETENESS.OF.LISP:
(IMPLIES (AND (STATE ST)
              (TAPE TAPE)
              (TURING.MACHINE TM))
         (AND (IMPLIES (NOT (BTMP (EVAL (tmi.x ST TAPE)
                                        NIL
                                        (tmi.fa TM)
                                        N)))
                       (NOT (BTMP (TMI ST TAPE TM
                                       (tmi.k ST TAPE TM N)))))
              (IMPLIES (NOT (BTMP (TMI ST TAPE TM K)))
                       (EQUAL (TMI ST TAPE TM K)
                              (EVAL (tmi.x ST TAPE)
                                    NIL
                                    (tmi.fa TM)
                                    (tmi.n ST TAPE TM K)))))),
```

Before proving this theorem we must define the functions tmi.x, tmi.fa, tmi.k and tmi.n.

We find the above formal statement of the completeness result to be intuitively appealing. However, the reader should study it to confirm that it indeed captures the intuitive notion. Some earlier formalizations by us were in fact inadequate. For example, one might wonder why we talk of the correspondences set up by tmi.x and tmi.fa instead of stating part (b) simply as:

> For every Turing machine TM, there exists a function alist fa,
> such that for every state ST and tape TAPE, there exists an x

such that:
if the computation of TM (starting in ST) on TAPE
terminates with tape t then the computation of x under
NIL and fa terminates with t.

However, observe that the existentially quantified x -- the expression EVAL is to evaluate -- can be chosen after TM, ST, and TAPE are fixed. Thus, a suitable x could be constructed by running the ideal Turing machine interpreter on the given TM, ST, and TAPE until it halts (as it must do by hypothesis), get the final tape, and embed it in a QUOTE form. Thus, when EVAL evaluates x it will return the final tape. Indeed, an EVAL that only knew how to interpret QUOTE would satisfy part (b) of the above statement of Turing completeness! Rearranging the quantifiers provides no relief. If the quantification on fa is made the innermost, one can appeal to a similar trick and embed the correct final answer in a QUOTE form inside the definition of a single dummy function.

Our statement does not admit such short cuts. Since tmi.fa is a function only of TM, one must invent a suitable LISP program from TM without knowing what its input will be. Since tmi.x is a function only of ST and TAPE, one must invent a suitable input for the LISP program without knowing which Turing machine will run it.

Proving the Turing completeness of EVAL is not entirely academic. The definition of EVAL requires about two pages. Its correctness as a formalization of Pure LISP is not nearly as obvious as the correctness of our formalization of Turing machines, which requires only half a page of definitions. The completeness result will assure us that EVAL — whether a formalization of LISP or not -- is as powerful as the Turing machine paradigm.

For our proof of the Turing completeness of LISP we define (tmi.fa TM) to be a function alist that contains a LISP encoding of the ideal Turing machine interpreter together with a single dummy function that returns the Turing

machine to be interpreted. Our definition of (tmi.x ST TAPE) returns the LISP s-expression that calls (the LISP version of) the ideal Turing machine interpreter on ST, TAPE, and the Turing machine TM. The heart of our proof is a "correspondence lemma" establishing that the LISP version of the ideal Turing machine interpreter behaves just like the ideal interpreter.

The remainder of this document presents formally the ideas sketched informally above. In the following sections we define Turing machines in our logic and develop the completeness proof. We simultaneously present the input necessary to lead our mechanical theorem-prover to the proof. Indented formulas preceded by the words "Definition," "Theorem," or "Disable" are actually commands to the theorem-prover to define functions, prove theorems, or cease to use certain facts. During our discussion we so present every command used to lead the system to the completeness proof, starting from the library of lemmas and definitions produced by our proof of the unsolvability of the halting problem [3].

5. The Formalization of Turing Machines

We here describe we how we have formalized in our logic the description of Turing machines in [4].

A tape is a sequence of 0's and 1's, containing a finite number of 1's, together with some pointer that indicates which cell of the tape the machine is "on." The symbol in that cell (a 0 or a 1) is the "current symbol."

We represent a tape by a pair. The CAR of the pair is a list of 0's and 1's representing the left half of the tape, in reverse order. The CDR of the pair a list of 0's and 1's representing the right half of the tape. The current symbol is the first one on the right half of the tape. To represent the infinite half-tape of 0's we use 0, since both the CAR and the CDR of 0 is 0. The function TAPE, defined below, returns T or F according to whether its

argument is a Turing machine tape.

```
Definition.
(SYMBOL X) = (MEMBER X '(0 1))

Definition.
(HALF.TAPE X)
   =
(IF (NLISTP X)
    (EQUAL X 0)
    (AND (SYMBOL (CAR X))
         (HALF.TAPE (CDR X))))

Definition.
(TAPE X)
   =
(AND (LISTP X)
     (HALF.TAPE (CAR X))
     (HALF.TAPE (CDR X)))
```

For example, to represent the tape below (containing all 0's to the extreme left and right) with the machine on the cell indicated by the "M"

```
                        M
... 0 1 0 1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 1 1 0 ...
```

we can use the following:

```
'((1 1 1 0 1 1 0 1 . 0) . (0 1 1 1 1 0 1 1 1 1 1 . 0))
```

The operations on a tape are to move to the left, move to the right, or replace the current symbol with a 0 or a 1. The reader should briefly contemplate the formal transformations induced by these operations. For example, if the current tape is given by (CONS left (CONS s right)) then the result of moving to the right is given by (CONS (CONS s left) right).

A Turing machine is a list of 4-tuples which, given an initial state and tape, uniquely determines a succession of operations and state changes. If

the current state is q and the current symbol is s and the machine contains a four-tuple (q s op q'), then q' becomes the new current state after the tape is modified as described by op; op can be L, which means move left, R, which means move right, or a symbol to write on the tape at the current position. If there is no matching four-tuple in the machine, the machine halts, returning the tape. The function TURING.MACHINE, defined below, returns T or F according to whether its argument is a well-formed Turing machine:

```
Definition.
(OPERATION X) = (MEMBER X '(L R 0 1))


Definition.
(STATE X) = (LITATOM X)


Definition.
(TURING.4TUPLE X)
   =
(AND (LISTP X)
     (STATE (CAR X))
     (SYMBOL (CADR X))
     (OPERATION (CADDR X))
     (STATE (CADDDR X))
     (EQUAL (CDDDDR X) NIL))


Definition.
(TURING.MACHINE X)
   =
(IF (NLISTP X)
    (EQUAL X NIL)
    (AND (TURING.4TUPLE (CAR X))
         (TURING.MACHINE (CDR X))))
```

An example Turing machine in this representation is:

```
tm:  '((Q0 1 0 Q1)
       (Q1 0 R Q2)
       (Q2 1 0 Q3)
       (Q3 0 R Q4)
       (Q4 1 R Q4)
       (Q4 0 R Q5)
       (Q5 1 R Q5)
       (Q5 0 1 Q6)
       (Q6 1 R Q6)
       (Q6 0 1 Q7)
       (Q7 1 L Q7)
       (Q7 0 L Q8)
       (Q8 1 L Q1)
       (Q1 1 L Q1))
```

This machine is the one shown on page 14 of [4].

Our Turing machine interpreter, TMI, is defined in terms of two other defined functions. The first, INSTR, searches up the Turing machine description for the current state and symbol. If it finds them, it returns the pair whose CAR is the operation and whose CADR is the new state. Otherwise, it returns F. The second function, NEW.TAPE, takes the operation and the current tape and returns the new tape configuration.

```
Definition.
(INSTR ST SYM TM)
   =
(IF (LISTP TM)
    (IF (EQUAL ST (CAAR TM))
        (IF (EQUAL SYM (CADAR TM))
            (CDDAR TM)
            (INSTR ST SYM (CDR TM)))
        (INSTR ST SYM (CDR TM)))
    F)


Definition.
(NEW.TAPE OP TAPE)
   =
(IF (EQUAL OP 'L)
    (CONS (CDAR TAPE)
          (CONS (CAAR TAPE) (CDR TAPE)))
    (IF (EQUAL OP 'R)
        (CONS (CONS (CADR TAPE) (CAR TAPE))
              (CDDR TAPE))
        (CONS (CAR TAPE)
              (CONS OP (CDDR TAPE)))))
```

Our Turing machine interpreter is defined as follows:

```
Definition.
(TMI ST TAPE TM N)
   =
(IF (ZEROP N)
    (BTM)
    (IF (INSTR ST (CADR TAPE) TM)
        (TMI (CADR (INSTR ST (CADR TAPE) TM))
             (NEW.TAPE (CAR (INSTR ST (CADR TAPE) TM))
                       TAPE)
             TM
             (SUB1 N))
        TAPE))
```

We now illustrate TMI. Let tm be the example Turing machine shown above. The partial recursive function computed by the machine is that defined by f(x)=2x.

Let tape be the tape '(0 . (1 1 1 1 1 . 0)), in which the current cell is the leftmost 1 in a block of 4+1 consecutive 1's. Then the following is a theorem:

```
(TMI 'Q0 tape tm N)
   =
(IF (LESSP N 78)
    (BTM)
    '((0 0 0 0 . 0) . (0 0 1 1 1 1 1 1 1 1 . 0)))
```

Observe that there are 8 1's on the final tape.

Our TMI differs from Roger's interpreter in the following minor respects. Rogers uses B (blank) instead of our 0. In his description of the interpreter, Rogers permits tapes containing infinitely many 1's. But his characterization of partial recursive functions does not use this fact and we have limited our machines to tapes containing only a finite number of 1's. Finally, Rogers' interpreter does certain kinds of syntax checking (e.g., insuring that the op is either L, R, 0, or 1) that ours does not do, with the result that our interpreter will execute more machines on more tapes than

Rogers'.

6. The Definitions of tmi.fa, tmi.x, tmi.n, and tmi.k

In this section we define the LISP analogues of INSTR, NEW.TAPE, and TMI, and we then define tmi.fa, tmi.x, tmi.n, and tmi.k.

To define a LISP program on the function alist argument of EVAL one adds to the alist a pair of the form (name . (args body)), where name is the name of the program, args is a list of the formal parameters, and body is the s-expression describing the computation to be performed. We wish to define the LISP counterpart of the ideal Turing machine interpreter. We will call the program 'TMI.[1] It is defined in terms of two other programs, 'INSTR and 'NEW.TAPE. The three functions defined below return as their values the (args body) part of the LISP definitions.

```
Definition.
(INSTR.DEFN)
   =
'((ST SYM TM)
  (IF (LISTP TM)
      (IF (EQUAL ST (CAR (CAR TM)))
          (IF (EQUAL SYM (CAR (CDR (CAR TM))))
              (CDR (CDR (CAR TM)))
              (INSTR ST SYM (CDR TM)))
          (INSTR ST SYM (CDR TM)))
      F))
```

[1]We refer to LISP programs by the literal atom used to name them. Thus 'TMI refers to the LISP program of that name, while TMI refers to the function.

```
Definition.
(NEW.TAPE.DEFN)
   =
'((OP TAPE)
  (IF (EQUAL OP (QUOTE L))
      (CONS (CDR (CAR TAPE))
            (CONS (CAR (CAR TAPE)) (CDR TAPE)))
      (IF (EQUAL OP (QUOTE R))
          (CONS (CONS (CAR (CDR TAPE)) (CAR TAPE))
                (CDR (CDR TAPE)))
          (CONS (CAR TAPE)
                (CONS OP (CDR (CDR TAPE)))))))


Definition.
(TMI.DEFN)
   =
'((ST TAPE TM)
  (IF (INSTR ST (CAR (CDR TAPE)) TM)
      (TMI (CAR (CDR (INSTR ST (CAR (CDR TAPE)) TM)))
           (NEW.TAPE (CAR (INSTR ST (CAR (CDR TAPE)) TM))
                     TAPE)
           TM)
      TAPE))
```

Observe that the LISP program 'TMI takes only the three arguments ST, TAPE, and TM; it does not take the number of instructions to execute since that number is not available to programs. Thus, the program 'TMI is the LISP analogue of the ideal Turing machine interpreter, rather than our function TMI.

```
Definition.
(KWOTE X)
   =
(LIST 'QUOTE X)
```

(KWOTE 'A) is '(QUOTE A). In general (KWOTE X) returns a LISP form that when evaluated returns X.

Here now is the function tmi.fa that when given a Turing machine returns the corresponding LISP program environment:

```
Definition.
(tmi.fa TM)
   =
(LIST (LIST 'TM NIL (KWOTE TM))
      (CONS 'INSTR (INSTR.DEFN))
      (CONS 'NEW.TAPE (NEW.TAPE.DEFN))
      (CONS 'TMI (TMI.DEFN)))
```

Note that (tmi.fa TM) is a function alist containing definitions for 'INSTR, 'NEW.TAPE, and 'TMI, as well as the dummy program 'TM which, when called, returns the Turing machine TM to be interpreted. Thus, the Turing machine is available during the LISP evaluation of the s-expression returned by tmi.x. But it is not available to tmi.x itself.

Here is the function that, when given a Turing machine state and tape, returns the corresponding input for the LISP program 'TMI.

```
Definition.
(tmi.x ST TAPE)
   =
(LIST 'TMI
      (KWOTE ST)
      (KWOTE TAPE)
      '(TM))
```

The form returned by tmi.x is a call of (the LISP version of) the ideal interpreter, 'TMI, on arguments that evaluate to ST, TAPE, and TM.

We now define the tmi.k, which is supposed to return a number of Turing machine steps sufficient to insure that TMI terminates gracefully if EVAL terminates in stack space N.

```
Definition.
(tmi.k ST TAPE TM N)
   =
(DIFFERENCE N (ADD1 (LENGTH TM)))
```

Finally, we define tmi.n, which returns a stack depth sufficient to insure

that EVAL terminates gracefully if TMI terminates in K steps.

```
Definition.
(tmi.n ST TAPE TM K)
   =
(PLUS K (ADD1 (LENGTH TM)))
```

We have now completed all the definitions necessary to formally state TURING.COMPLETENESS.OF.LISP. The reader may now wish to prove the theorem himself.

7. The Formal Proof

7.1. Some Preliminary Remarks about BTM

We wish to establish a correspondence between calls of the LISP program 'TMI and the function TMI. We will do so by first noting the correspondence between primitive LISP expressions, such as '(CAR (CDR X)), and their logical counterparts, e.g., (CAR (CDR X)). These primitive correspondences are explicit in the definition of EVAL. We then prove correspondence lemmas for the defined LISP programs 'INSTR and 'NEW.TAPE, before finally turning to 'TMI.

Unfortunately, the correspondence, even at the primitive level, is not as neat as one might wish because of the role of (BTM) in the LISP computations. For example, the value of the LISP program 'CDR applied to some object is the CDR of the object provided the object is not (BTM). Thus, if the LISP variable symbol 'X is bound to some object, X, then the value of the LISP expression '(CAR (CDR X)) is (CAR (CDR X)) only if X is not (BTM), its CDR is not (BTM) and the CAR of that is not (BTM). If 'X were bound to the pair (CONS 3 (BTM)), then '(CAR (CDR X)) would evaluate to (BTM) while (CAR (CDR X)) would be 0, since the CAR of the nonlist (BTM) is 0. LISP programs cannot construct an object like (CONS 3 (BTM)), for if a LISP program tried to 'CONS

something onto (BTM) the result would be (BTM). But such "uncomputable" objects can be constructed in the logic and might conceivably be found in the input to EVAL.

Therefore, the correspondence lemmas for our programs 'INSTR, 'NEW.TAPE, and 'TMI contain conditions that prohibit the occurrence of (BTM) within the objects to which the programs are applied. These conditions are stated with the function CNB (read "contains no (BTM)'s"). To use such correspondence lemmas when we encounter calls of these programs we must be able to establish that their arguments contain no (BTM)'s. This happens to be the case because in the main theorem 'TMI is called on a STATE, TAPE, and TURING.MACHINE and all subsequent calls are on components of those objects, none of which contains a (BTM).

The correspondence lemmas for recursive programs are complicated in a much more insidious way by the possibility of nontermination or mere "stack overflow." Consider an arbitrary call of 'INSTR:

(LIST 'INSTR st sym tm),

where st, svm and tm are arbitrary LISP expressions. Suppose st, sym, and tm evaluate, respectively, to st, sym, and tm and suppose further that st, sym, and tm contain no (BTM)'s. Is the value of (LIST 'INSTR st sym tm) equal to (INSTR st sym tm)? It depends on the amount of stack depth available. If the depth exceeds the length of tm, |tm|, then the answer computed is (INSTR st sym tm). If the depth is less than or equal to |tm| is the answer computed (BTM)? Not necessarily. It depends on where (and whether) the 4-tuple for st and sym occurs in tm. If it occurs early enough, 'INSTR will compute the same answer as INSTR even with insufficient stack space to explore the entire tm.

In stating the correspondence lemma for 'INSTR we could add a hypothesis that the amount of stack depth available exceeds |tm|. However, not all calls of 'INSTR arising from the computation described in the main theorem provide

adequate stack space. Part (a) of the main theorem forces us to deal explicitly with the conditions under which 'TMI computes (BTM). Thus it is necessary to know not merely when when 'INSTR computes INSTR but also when 'INSTR computes (BTM).

We therefore will define a "derived function," INSTRN, in the logic that is like INSTR except that it takes a stack depth argument in addition to its other arguments and returns (BTM) when it exhausts the stack, just as does the evaluation of 'INSTR. We will then prove a correspondence lemma between 'INSTR and INSTRN. We will take a similar approach to 'TMI, defining the derived function TMIN that takes a stack depth and uses INSTRN where TMI uses INSTR. After proving the correspondence lemma between 'TMI and TMIN we will have eliminated all consideration of EVAL from the problem at hand and then focus our attention on the relation between TMIN and TMI.

At this point in the mechanical proofs we defined "contains no (BTM)'s" and had the machine prove a variety of general purpose lemmas about arithmetic, EVAL, and CNB. These details are merely distracting to the reader, as they have nothing to do with Turing machines. We have therefore put those events in the appendix of this document.

7.2. The Correspondence Lemma for 'INSTR

```
Definition.
(INSTRN ST SYM TM N)
   =
(IF (ZEROP N)
    (BTM)
    (IF (LISTP TM)
        (IF (EQUAL ST (CAAR TM))
            (IF (EQUAL SYM (CADAR TM))
                (CDDAR TM)
                (INSTRN ST SYM (CDR TM) (SUB1 N)))
            (INSTRN ST SYM (CDR TM) (SUB1 N)))
        F))
```

The proof of the correspondence between 'INSTR and INSTRN requires a rather

odd induction, in which certain variables are replaced by constants. We have to tell the theorem-prover about this induction by defining a recursive function that mirrors the induction we desire.

```
Definition.
(EVAL.INSTR.INDUCTION.SCHEME st sym tm VA TM N)
   =
(IF (ZEROP N)
    T
    (EVAL.INSTR.INDUCTION.SCHEME
         'ST
         'SYM
         '(CDR TM)
         (LIST (CONS 'ST (EVAL st VA (tmi.fa TM) N))
               (CONS 'SYM
                     (EVAL sym VA (tmi.fa TM) N))
               (CONS 'TM
                     (EVAL tm VA (tmi.fa TM) N)))
         TM
         (SUB1 N)))
```

Here is the correspondence lemma for 'INSTR. Note that in the "Hint" we explicitly tell the machine which induction to use.

```
Theorem.  EVAL.INSTR (rewrite):
(IMPLIES
     (AND (CNB (EV 'AL st VA (tmi.fa TM) N))
          (CNB (EV 'AL sym VA (tmi.fa TM) N))
          (CNB (EV 'AL tm VA (tmi.fa TM) N)))
     (EQUAL (EV 'AL (LIST 'INSTR st sym tm)
                    VA
                    (tmi.fa TM)
                    N)
            (INSTRN (EV 'AL st VA (tmi.fa TM) N)
                    (EV 'AL sym VA (tmi.fa TM) N)
                    (EV 'AL tm VA (tmi.fa TM) N)
                    N)))
Hint:  Induct as for
       (EVAL.INSTR.INDUCTION.SCHEME st sym tm VA TM N).
```

Since the correspondence lemmas are the heart of the proof, we will dwell on this one briefly.

As noted, an arbitrary call of 'INSTR has the form (LIST 'INSTR st sym tm), where st, sym and tm are arbitrary LISP expressions. Suppose the call is evaluated in an arbitrary variable alist, VA, the function alist (tmi.fa TM), and with an arbitrary stack depth N available. Let st, sym, and tm be the results of evaluating the expressions st, sym, and tm respectively. Then the arbitrary call of 'INSTR returns (INSTRN st sym tm N) provided only that st, sym, and tm contain no (BTM)'s.

The reader of [3] will recall that (EVAL X VA FA N) is defined as (EV 'AL X VA FA N), since EVAL and EVLIST are mutually recursive. We state our lemmas in terms of EV so they will be more useful as rewrite rules. (Expressions beginning with EVAL will be expanded to EV expressions by the definition of EVAL, and then our lemmas will apply.)

The function alist for the 'INSTR correspondence lemma need not be (tmi.fa TM) for the lemma to hold; it is sufficient if the function alist merely contains the expected definition for 'INSTR. However, as (tmi.fa TM) is the only function alist we will need in this problem it was simplest to state the lemma this way.

We invite the reader to do the theorem-prover's job and construct the proof of the lemma from the foregoing lemmas and definitions.

Once proved, this lemma essentially adds 'INSTR to the set of "primitive" LISP programs with logical counterparts. That is, when the theorem-prover sees:

```
(EVAL '(INSTR ST (CAR (CDR TAPE)) TM) va (tmi.fa TM) N),
```

as it will when working with the body of 'TMI, it will replace it by:

```
(INSTRN ST (CAR (CDR TAPE)) TM N)
```

provided 'ST, 'SYM, and 'TM are bound in va to ST, SYM, and TM (respectively) and ST, SYM, and TM contain no (BTM)'s. This eliminates a call of EVAL.

7.3. The Correspondence Lemma for 'NEW.TAPE

```
Theorem.  EVAL.NEW.TAPE (rewrite):
(IMPLIES
 (AND (CNB (EV 'AL op VA (tmi.fa TM) N))
      (CNB (EV 'AL tape VA (tmi.fa TM) N)))
 (EQUAL (EV 'AL (LIST 'NEW.TAPE op tape)
                VA
                (tmi.fa TM)
                N)
        (IF (ZEROP N)
            (BTM)
            (NEW.TAPE (EV 'AL op VA (tmi.fa TM) N)
                      (EV 'AL tape VA (tmi.fa TM) N))))))
```

7.4. The Correspondence Lemma for 'TMI

We now consider the correspondence lemma for 'TMI. As we did for 'INSTR, consider an arbitrary call of 'TMI, (LIST 'TMI st tape tm). Let st, tape, and tm be the values of the st, tape, and tm expressions. Suppose the stack depth available is N. Then at first glance, the evaluation of (LIST 'TMI st tape tm) is just (TMI st tape tm N), since 'TMI causes EVAL to recurse down the stack exactly once for every time TMI decrements its step count N. But again we have forgotten 'INSTR and the possibility that N is sufficiently small that 'INSTR will return (BTM) while trying to fetch the next op and state. We therefore define the derived function TMIN, which is just like TMI except that instead of using INSTR it uses INSTRN. We then prove that the correspondence lemma between 'TMI and TMIN.

Before we proceed we prove two simple lemmas about INSTRN and NEW.TAPE.

```
Theorem.  CNB.INSTRN (rewrite):
(IMPLIES (AND (NOT (BTMP (INSTRN ST SYM TM N)))
              (CNB TM))
         (CNB (INSTRN ST SYM TM N)))
```

```
Theorem.  CNB.NEW.TAPE (rewrite):
(IMPLIES (AND (CNB OP) (CNB TAPE))
         (CNB (NEW.TAPE OP TAPE)))
```

```
Disable NEW.TAPE.
```

When we disable a function name we tell the theorem-prover that future proofs about the function do not require knowledge of the definition.

Here is the derived function for the program 'TMI.

```
Definition.
(TMIN ST TAPE TM N)
   =
(IF (ZEROP N)
    (BTM)
    (IF (BTMP (INSTRN ST (CADR TAPE) TM (SUB1 N)))
        (BTM)
        (IF (INSTRN ST (CADR TAPE) TM (SUB1 N))
            (TMIN (CADR (INSTRN ST (CADR TAPE) TM (SUB1 N)))
                  (NEW.TAPE
                    (CAR (INSTRN ST (CADR TAPE) TM (SUB1 N)))
                    TAPE)
                  TM
                  (SUB1 N))
            TAPE)))
```

The induction scheme we use for the 'TMI correspondence lemma is suggested by the following function:

```
Definition.
(EVAL.TMI.INDUCTION.SCHEME st tape tm VA TM N)
   =
(IF (ZEROP N)
    T
    (EVAL.TMI.INDUCTION.SCHEME
     '(CAR (CDR (INSTR ST (CAR (CDR TAPE)) TM)))
     '(NEW.TAPE (CAR (INSTR ST (CAR (CDR TAPE)) TM))
                TAPE)
     'TM
     (LIST (CONS 'ST
                 (EV 'AL st VA (tmi.fa TM) N))
           (CONS 'TAPE
                 (EV 'AL tape VA (tmi.fa TM) N))
           (CONS 'TM
                 (EV 'AL tm VA (tmi.fa TM) N)))
     TM
     (SUB1 N)))
```

Here is the correspondence lemma for 'TMI, together with the hint for how to prove it.

```
Theorem.  EVAL.TMI (rewrite):
(IMPLIES
      (AND (CNB (EV 'AL st VA (tmi.fa TM) N))
           (CNB (EV 'AL tape VA (tmi.fa TM) N))
           (CNB (EV 'AL tm VA (tmi.fa TM) N)))
      (EQUAL (EV 'AL
                 (LIST 'TMI st tape tm)
                 VA
                 (tmi.fa TM)
                 N)
             (TMIN (EV 'AL st VA (tmi.fa TM) N)
                   (EV 'AL tape VA (tmi.fa TM) N)
                   (EV 'AL tm VA (tmi.fa TM) N)
                   N)))
Hint:  Induct as for
       (EVAL.TMI.INDUCTION.SCHEME st tape tm VA TM N).
```

7.5. EVAL of tmi.x

EVAL occurs twice in the completeness theorem. The two occurrences are:

```
(EVAL (tmi.x ST TAPE) NIL (tmi.fa TM) N)
```

and

```
(EVAL (tmi.x ST TAPE) NIL (tmi.fa TM) (tmi.n ST TAPE TM K)).
```

But since (tmi.x ST TAPE) is just

```
(LIST 'TMI (LIST 'QUOTE ST) (LIST 'QUOTE TAPE) '(TM)).
```

we can eliminate both uses of EVAL from the main theorem by appealing to the correspondence lemma for 'TMI.

```
Theorem.  EVAL.tmi.x (rewrite):
(IMPLIES (AND (CNB ST) (CNB TAPE) (CNB TM))
         (EQUAL (EV 'AL
                    (tmi.x ST TAPE)
                    NIL
                    (tmi.fa TM)
                    N)
                (IF (ZEROP N)
                    (BTM)
                    (TMIN ST TAPE TM N))))
```

The test on whether N is 0 is necessary to permit us to use the correspondence lemma for 'TMI. If N=0 then the evaluation of the third argument in the call of 'TMI returns (BTM) and we cannot relieve the hypothesis that the value of the actual expression contains no (BTM).

Disable tmi.x.

7.6. The Relation Between TMI and TMIN

By using the lemma just proved the main theorem becomes:

```
(IMPLIES
 (AND (STATE ST)
      (TAPE TAPE)
      (TURING.MACHINE TM))
 (AND
  (IMPLIES (AND (NOT (ZEROP N))
                (NOT (BTMP (TMIN ST TAPE TM N))))
           (NOT (BTMP (TMI ST TAPE TM
                           (tmi.k ST TAPE TM N)))))
  (IMPLIES (NOT (BTMP (TMI ST TAPE TM K)))
           (EQUAL (TMI ST TAPE TM K)
                  (TMIN ST TAPE TM (tmi.n ST TAPE TM K))))))
```

That is, EVAL has been eliminated and we must prove two facts relating TMIN and TMI. We seek a general statement of the relation between TMI and TMIN. It turns out there is a very simple one. Given certain reasonable conditions on the structure of TM,

(TMI ST TAPE TM K) = (TMIN ST TAPE TM (PLUS K (ADD1 (LENGTH TM)))).

Note that once this result has been proved, the main theorem follows from the definitions of tmi.k and tmi.n. But let us briefly consider why this result is valid.

Clearly, if (TMI ST TAPE TM K) terminates normally within K steps, then TMIN will also terminate with the same answer with K+|TM|+1 stack depth. The reason TMIN needs more stack depth than TMI needs steps is that on the last step, TMIN will need enough space to run INSTRN all the way down TM to confirm

that the current state is a final state. Of course, we use the facts that, given enough stack space, INSTRN is just INSTR and does not return (BTM):

```
Theorem.  INSTRN.INSTR (rewrite):
(IMPLIES (LESSP (LENGTH TM) N)
         (EQUAL (INSTRN ST SYM TM N)
                (INSTR ST SYM TM)))
```

```
Theorem.  NBTMP.INSTR (rewrite):
(IMPLIES (CNB TM)
         (NOT (BTMP (INSTR ST SYM TM))))
```

It is less clear that if TMI returns (BTM) after executing K steps, then the TMIN expression also returns (BTM), even though its stack depth is K+|TM|+1. What prevents TMIN, after recursing K times to take the first K steps, from using some of the extra stack space to take a few more steps, find a terminal state, and halt gracefully? In fact, TMIN may well take a few more steps (if the state transitions it runs through occur early in the TM so that it does not exhaust its stack looking for them). However, to confirm that it has reached a terminal state it must make an exhaustive sweep through the TM, which will of necessity use up all the stack space. Another way of putting it is this: if TMIN is to terminate gracefully, INSTRN must return F, but INSTRN will not return F unless given a stack depth greater than |TM|.

```
Theorem.  INSTRN.NON.F (rewrite):
(IMPLIES (AND (TURING.MACHINE TM)
              (LEQ N (LENGTH TM)))
         (INSTRN ST SYM TM N))
```

We therefore conclude:

```
Theorem.  TMIN.BTM (rewrite):
(IMPLIES (AND (TURING.MACHINE TM)
              (LEQ N (LENGTH TM)))
         (EQUAL (TMIN ST TAPE TM N) (BTM)))
```

So it is easy now to prove:

```
Theorem.  TMIN.TMI (rewrite):
(IMPLIES (TURING.MACHINE TM)
         (EQUAL (TMI ST TAPE TM K)
                (TMIN ST TAPE TM
                      (PLUS K (ADD1 (LENGTH TM))))))
```

The reader is reminded that the lemmas being displayed here are not merely read by the theorem-prover. They are proved.

7.7. Relating CNB to Turing Machines

All that remains is to establish the obvious connections between the predicates STATE, TAPE, and TURING.MACHINE and the concept of "contains no (BTM)'s."

```
Theorem.  SYMBOL.CNB (rewrite):
(IMPLIES (SYMBOL SYM) (CNB SYM))

Disable SYMBOL.

Theorem.  HALF.TAPE.CNB (rewrite):
(IMPLIES (HALF.TAPE X) (CNB X))

Theorem.  TAPE.CNB (rewrite):
(IMPLIES (TAPE X) (CNB X))

Disable TAPE.

Theorem.  OPERATION.CNB (rewrite):
(IMPLIES (OPERATION OP) (CNB OP))

Disable OPERATION.

Theorem.  TURING.MACHINE.CNB (rewrite):
(IMPLIES (TURING.MACHINE TM) (CNB TM))

Disable TURING.MACHINE.
```

7.8. The Main Theorem

We are now able to prove that LISP is Turing complete.

```
Theorem.  TURING.COMPLETENESS.OF.LISP:
(IMPLIES
 (AND (STATE ST)
      (TAPE TAPE)
      (TURING.MACHINE TM))
 (AND
  (IMPLIES (NOT (BTMP (EVAL (tmi.x ST TAPE)
                            NIL
                            (tmi.fa TM)
                            N)))
           (NOT (BTMP (TMI ST TAPE TM
                           (tmi.k ST TAPE TM N)))))
  (IMPLIES (NOT (BTMP (TMI ST TAPE TM K)))
           (EQUAL (TMI ST TAPE TM K)
                  (EVAL (tmi.x ST TAPE)
                        NIL
                        (tmi.fa TM)
                        (tmi.n ST TAPE TM K))))))
```

Q.E.D.

8. Conclusion

We have described a mechanically checked proof of the Turing completeness of Pure LISP. While the result is obvious to all LISP programmers, it was not obvious, even to us, that a mechanically checked proof could be constructed so easily.

It is unusual to see a completely formal characterization of any computational paradigm. We were particularly heartened to see how simply the Turing machine paradigm could be expressed formally in our constructive logic. Because we are constructivists at heart, we found the step counter intuitively appealing. Furthermore, it did not obstruct the proofs. In any case, one's formalism must be able to express the notion that the interpreter "runs forever." We invite adherents of other mechanized logical systems to construct and mechanically check the entire proof -- from the definition of EVAL and Turing machines through the completeness theorem -- for comparison

with this one.

Another aspect of the proof that delighted us was the fact that our Pure LISP interpreter — defined with its stack depth parameter to fit it into our constructive logic -- was used to simulate the ideal Turing machine interpreter, not our TMI with its step counter.

It should be noted that our use of the "derived functions" INSTRN and TMIN to factor EVAL out of the proof is very similar to McCarthy's notion of functional semantics. Our derived functions capture the input/output semantics of those programs in addition to such intensional properties as their termination and stack overflow properties. It would have been much harder to construct the final sequence of proofs (in which we argued about termination conditions for 'INSTR and 'TMI) had EVAL been involved.

Finally we would like to remind the reader of our conjecture that it may be practical, with the help of automatic theorem-provers, to do much mathematics formally.

Some readers may feel that the proof was presented at such a low level of detail that it could have been checked by a far less sophisticated theorem-prover. The reader is urged to review the 17 named theorems displayed in Section 7 and produce a proof of each. We stress the word "proof" because we have found that many people imagine that they have a proof of a formula when in fact all they have is a gut-level intuitive grasp of what the formula says and find themselves in agreement. We would be most interested if any reader successfully checks the entire proof here with another mechanical theorem-prover or proof checker.

Other readers may feel that the proof presented here was far from formal; that is true and exactly our point. The task required of the user in this proof was to describe the proof at roughly the level we are accustomed to

presenting proofs. It was the machine that constructed the formal proof.

Appendix A. Some Elementary Lemmas used in the Proof

After defining the functions tmi.fa, tmi.x, tmi.n, and tmi.k used in the statement of the completeness theorem, but before proving the first correspondence lemma, we had the theorem-prover do some preliminary work with arithmetic, EVAL, CNB, and tmi.fa. In all, 22 commands are listed in this appendix: 16 lemmas, 1 definition, and 5 disable commands. All of the results are elementary and are unconcerned with Turing machines themselves. Indeed, as the reader will see, the arithmetic, EVAL, and CNB results are of a completely general and basic nature and will be of use in other proofs about EVAL. The one lemma about tmi.fa is strictly unnecessary for the completeness proof, but makes the output less voluminous. Nevertheless, in the interests of providing a complete record of the completeness proof, we list the commands here.

I.1. Elementary Lemmas about Arithmetic

```
Theorem.  LENGTH.0 (rewrite):
(EQUAL (EQUAL (LENGTH X) 0)
       (NLISTP X))


Theorem.  PLUS.EQUAL.0 (rewrite):
(EQUAL (EQUAL (PLUS I J) 0)
       (AND (ZEROP I) (ZEROP J)))


Theorem.  PLUS.DIFFERENCE (rewrite):
(EQUAL (PLUS (DIFFERENCE I J) J)
       (IF (LEQ I J) (FIX J) I))


Disable DIFFERENCE.
```

I.2. Elementary Lemmas about EVAL

These lemmas all follow directly from the definition of EV. Indeed, taken together they are virtually equivalent to the definition. However, by making them available as rewrite rules we speed up the simplification of EV expressions in which the s-expression being evaluated is a constant. For example, after proving the lemmas, the EVAL expression

```
(EVAL '(IF (LISTP X)
           (CONS (CAR X) (CONS OP (CDR (CDR X))))
           F)
      (LIST (CONS 'X X) (CONS 'OP OP))
      FA
      N)
```

is immediately simplified to:

```
(IF (LISTP X)
    (CONS (CAR X) (CONS OP (CDR (CDR X))))
    F)
```

provided only that OP, X, (CAR X), (CDR X), and (CDDR X) are not (BTM). Were these lemmas not available as rewrite rules the EVAL expression would still simplify as above, but it would take longer as it would involve the heuristics controlling the opening up (many times) of the recursive function EV.

```
Theorem.  EVAL.FN.0 (rewrite):
(IMPLIES (AND (ZEROP N)
              (NOT (EQUAL FN 'QUOTE))
              (NOT (EQUAL FN 'IF))
              (NOT (SUBRP FN))
              (EQUAL VARGS
                     (EV 'LIST ARGS VA FA N)))
         (EQUAL (EV 'AL (CONS FN ARGS) VA FA N)
                (BTM)))
```

This lemma says that the value of an arbitrary LISP call of the program FN on a list of expressions, ARGS, is (BTM) if the stack depth is 0 and FN is not one of the LISP primitives.

```
Theorem.  EVAL.FN.1 (rewrite):
(IMPLIES
 (AND (NOT (ZEROP N))
      (NOT (EQUAL FN 'QUOTE))
      (NOT (EQUAL FN 'IF))
      (NOT (SUBRP FN))
      (EQUAL VARGS
             (EV 'LIST ARGS VA FA N)))
 (EQUAL
    (EV 'AL (CONS FN ARGS) VA FA N)
    (IF (BTMP VARGS)
        (BTM)
        (IF (BTMP (GET FN FA))
            (BTM)
            (EV 'AL
                (CADR (GET FN FA))
                (PAIRLIST (CAR (GET FN FA)) VARGS)
                FA
                (SUB1 N))))))
```

This lemma characterizes the result of an arbitrary LISP call of a nonprimitive program FN on ARGS when the stack depth is non-0. If either the evaluation of ARGS produces (BTM) or no definition for FN is found in the function alist, the result is (BTM). Otherwise, the result is obtained by evaluating the body of FN in an environment in which its formal parameters are bound to the values of the actuals and the stack depth is decremented by one.

The remaining EV lemmas characterize the behavior of EVAL on the primitive programs (SUBRs, IF, QUOTE, and atomic symbols) and characterize EVLIST on NIL and the general nonempty list.

```
Theorem.  EVAL.SUBRP (rewrite):
(IMPLIES (AND (SUBRP FN)
              (EQUAL VARGS
                     (EV 'LIST ARGS VA FA N)))
         (EQUAL (EV 'AL (CONS FN ARGS) VA FA N)
                (IF (BTMP VARGS)
                    (BTM)
                    (APPLY.SUBR FN VARGS))))
```

```
Theorem.  EVAL.IF (rewrite):
(IMPLIES
     (EQUAL VX1 (EV 'AL X1 VA FA N))
     (EQUAL (EV 'AL (LIST 'IF X1 X2 X3) VA FA N)
            (IF (BTMP VX1)
                (BTM)
                (IF VX1
                    (EV 'AL X2 VA FA N)
                    (EV 'AL X3 VA FA N)))))


Theorem.  EVAL.QUOTE (rewrite):
(EQUAL (EV 'AL (LIST 'QUOTE X) VA FA N)
       X)


Theorem.  EVAL.NLISTP (rewrite):
(AND (EQUAL (EV 'AL 0 VA FA N) 0)
     (EQUAL (EV 'AL (ADD1 N) VA FA N)
            (ADD1 N))
     (EQUAL (EV 'AL (PACK X) VA FA N)
            (IF (EQUAL (PACK X) 'T)
                T
                (IF (EQUAL (PACK X) 'F)
                    F
                    (IF (EQUAL (PACK X) NIL)
                        NIL
                        (GET (PACK X) VA))))))


Theorem.  EVLIST.NIL (rewrite):
(EQUAL (EV 'LIST NIL VA FA N) NIL)


Theorem.  EVLIST.CONS (rewrite):
(IMPLIES
   (AND (EQUAL VX (EV 'AL X VA FA N))
        (EQUAL VL (EV 'LIST L VA FA N)))
   (EQUAL (EV 'LIST (CONS X L) VA FA N)
          (IF (BTMP VX)
              (BTM)
              (IF (BTMP VL) (BTM) (CONS VX VL)))))
```

We now disable SUBRP and EV so that their definitions are not even considered. Such expressions as (SUBRP 'CONS) and (SUBRP 'TMI) will still rewrite to T and F respectively, because even disabled definitions are evaluated on explicit constants. EV expressions will henceforth be rewritten only by the lemmas above.

```
Disable SUBRP.

Disable EV.
```

I.3. CNB

```
Definition.
(CNB X)
   =
(IF (LISTP X)
    (AND (CNB (CAR X)) (CNB (CDR X)))
    (NOT (BTMP X)))
```

In the same spirit as our EVAL lemmas, we now prove a set of lemmas that let CNB expressions rewrite efficiently.

```
Theorem.  CNB.NBTM (rewrite):
(IMPLIES (CNB X) (NOT (BTMP X)))


Theorem.  CNB.CONS (rewrite):
(AND (EQUAL (CNB (CONS A B))
            (AND (CNB A) (CNB B)))
     (IMPLIES (CNB X) (CNB (CAR X)))
     (IMPLIES (CNB X) (CNB (CDR X))))


Theorem.  CNB.LITATOM (rewrite):
(IMPLIES (LITATOM X) (CNB X))


Theorem.  CNB.NUMBERP (rewrite):
(IMPLIES (NUMBERP X) (CNB X))


Disable CNB.
```

Note how these lemmas are used by the simplifier. Suppose the simplifier encounters the test (BTMP (CDR (CDR (CAR TM)))), as it does in the proof of the correspondence lemma for 'INSTR. Suppose the hypothesis of the conjecture being proved contains (CNB TM). Then the BTMP expression above is simplified to F by backwards chaining through the lemmas just proved:

```
(CNB TM)
  -> (CNB (CAR TM))
       -> (CNB (CDR (CAR TM)))
            -> (CNB (CDR (CDR (CAR TM))))
                 -> (BTMP (CDR (CDR (CAR TM)))) = F
```

I.4. Shutting Off Some Verbose Output

Consider (tmi.fa TM). If the constant functions it uses (INSTR.DEFN, NEW.TAPE.DEFN, and TMI.DEFN) are expanded -- as all constant functions are by the theorem-prover -- (tmi.fa TM) is a very large expression. Furthermore, the only use that is ever made of it is to use GET to fetch the definitions of those three programs, plus the definition of the dummy LISP program 'TM. Finally, (tmi.fa TM) occurs very frequently in the statement of our lemmas and even more frequently in their proofs. Rather than suffer through pages of output devoted to (tmi.fa TM), we here prove the four relevant properties of it, namely that GET returns the appropriate constant when given one of the known program names, and then disable tmi.fa so that it is no longer expanded into its verbose form. Again, these lemmas are not necessary to the proof, but they made the production of the proof much less taxing on the user.

```
Theorem.  GET.tmi.fa (rewrite):
(AND (EQUAL (GET 'TM (tmi.fa TM))
            (LIST NIL (KWOTE TM)))
     (EQUAL (GET 'INSTR (tmi.fa TM))
            (INSTR.DEFN))
     (EQUAL (GET 'NEW.TAPE (tmi.fa TM))
            (NEW.TAPE.DEFN))
     (EQUAL (GET 'TMI (tmi.fa TM))
            (TMI.DEFN)))


Disable tmi.fa.
```

REFERENCES

1. R. S. Boyer and J S. Moore. A Computational Logic. Academic Press, New York, 1979.

2. R. S. Boyer and J S. Moore. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures. In The Correctness Problem in Computer Science, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

3. R. S. Boyer and J S. Moore. A Mechanical Proof of the Unsolvability of the Halting Problem. Technical Report ICSA-CMP-28, University of Texas at Austin, 1982. To appear in the Journal of the Association for Computing Machinery.

4. H. Rogers, Jr.. Theory of Recursive Functions and Effective Computability. McGraw-Hill Book Company, New York, 1967.

DISTRIBUTION LIST

Defense Documentation Center (12 copies)
Cameron Station
Alexandria, VA 22314

Naval Research Laboratory (6 copies)
Technical Information Division
Code 2627
Washington, D.C. 20375

Office of Naval Research (2 copies)
Information Systems Program (437)
Arlington, VA 22217

Office of Naval Research
Code 200
Arlington, VA 22217

Office of Naval Research
Code 455
Arlington, VA 22217

Office of Naval Research
Code 458
Arlington, VA 22217

Office of Naval Research
Eastern/Central Regional Office
Bldg 114, Section D
666 Summer Street
Boston, MA 02210

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, IL 60605

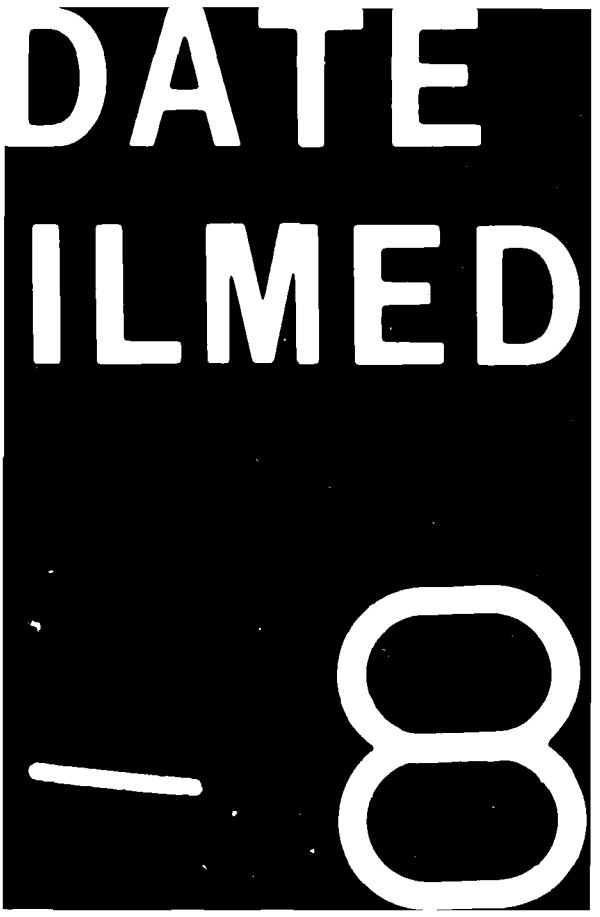
Office of Naval Research
Western Regional Office
1030 East Green Street
Pasadena, CA 91106

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
Code RD-1
Washington, D.C. 20380

Naval Ocean Systems Center
Advanced Software Technology Div.
Code 5200
San Diego, CA 92152

Mr. E. H. Gleissner
Naval Ship Research
& Development Center
Computation and Mathematics Dept.
Bethesda, MD 20084

Captain Grace M. Hopper (008)
Naval Data Automation Command
Washington Navy Yard
Building 166
Washington, D.C. 20374


DATE
ILMED
8